Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms

Jeonghun Cho Yunheung Paek [*]
Electrical Engineering & Computer Science Department
Korea Advanced Institute of Science & Technology
Daejon 305-701, Korea
{jhcho,ypaek}@soar.kaist.ac.kr

David Whalley [†]
Computer Science Department
Florida State University
Tallahassee, FL 32306-4530, USA
whalley@cs.fsu.edu

ABSTRACT

Finding an optimal assignment of program variables into registers and memory is prohibitively difficult in code generation for *application specific instruction-set processors* (ASIPs). This is mainly because, in order to meet stringent speed and power requirements for embedded applications, ASIPs commonly employ *non-orthogonal* architectures which are typically characterized by irregular data paths, heterogeneous registers and multiple memory banks. As a result, existing techniques mainly developed for relatively regular, orthogonal *general-purpose processors* (GPPs) are obsolete for these recently emerging ASIP architectures. In this paper, we attempt to tackle this issue by exploiting conventional *graph coloring* and *maximum spanning tree* (MST) algorithms with special constraints added to handle the non-orthogonality of ASIP architectures. The results in our study indicate that our algorithm finds a fairly good assignment of variables into heterogeneous registers and multi-memories while it runs extremely faster than previous work that employed exceedingly expensive algorithms to address this issue.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation/ compilers/optimization*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Parallel processors*

General Terms

Algorithms

[*]This work is supported in part by Brain Korea 21 project and Samsung Electronics Co.
[†]This work is supported in part by NSF contract EIA-9806525, CCR-9904943, and EIA-0072043.

Keywords

Compiler, dual memory, non-orthogonal architecture, memory assignment, graph coloring, and maximum spanning tree

1. INTRODUCTION

As embedded system designers strive to meet cost and performance goals demanded by the applications, the complexity of processors is ever increasingly optimized for certain application domains in embedded systems. Such optimizations need a process of *design space exploration* [8] to find hardware configurations that meet the design goals. The final configuration of a processor resulting from a design space exploration usually has an instruction set and the data path that are highly tuned for specific embedded applications. In this sense, they are collectively called *application specific instruction-set processors* (ASIPs).

An ASIP typically has a non-orthogonal architectre which can be characterized by irregular data paths containing *heterogeneous registers* and *multiple memory banks*. As an example of such an architecture, Figure 1 shows the Motorola DSP56000, a commercial off-the-shelf ASIP specifically designed for *digital signal processing* (DSP) applications. Note from the data path that the architecture lacks a large number of centralized general-purpose *homogeneous* registers; instead, it has multiple small register files where different files are distributed and dedicated to different sets of functional units. Also, note that it employs a multi-memory bank architecture which consists of program and data memory banks. In this architecture, two data memory banks are connected through two independent data buses, while a conventional von Neumann architecture has only a single memory bank. This type of memory architecture is supported by many embedded processors, such as Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP56000 and NEC uPD77016. One obvious advantage of this architecture is that it can access two data words in one instruction cycle.

Multi-memory bank architectures have been shown to be effective for many operations commonly found in embedded applications, such as N real multiplies:

$$z(i) = x(i) \times y(i) \quad i = 1, 2, ..., N$$

From this example, we can see that the application can operate at an ideal rate if a processor has two data memory banks

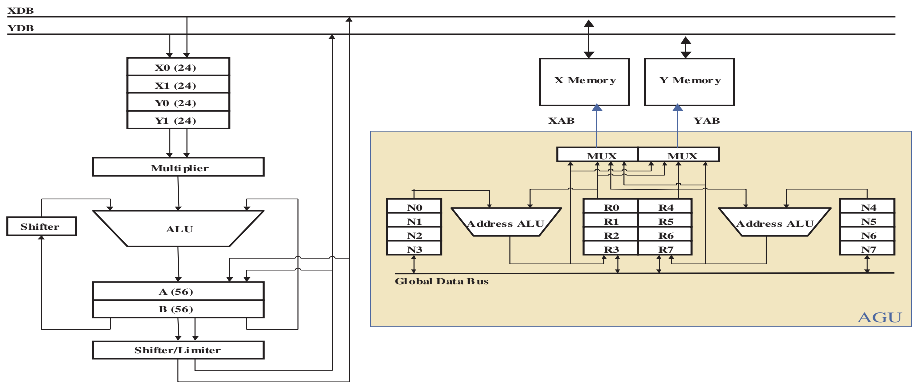


Figure 1: **Motorola DSP56000 data path with dual data memory banks X and Y**

so that two variables, $x(i)$ and $y(i)$, can be fetched simultaneously. But, we also can see that this ideal speed of operation is only possible with one condition: the variables should be assigned to different data memory banks. For instance, in the following DSP56000 assembly code implementing the N real multiplies, arrays x and y are assigned respectively to the two memory banks X and Y.

```
      move                 x:(r0)+,x0  y:(r4)+,y0
      mpyr    x0,y0,a      x:(r0)+,x0  y:(r4)+,y0
      do      #N-1,end
      mpyr    x0,y0,a      a,x:(r1)+   y:(r4)+,y0
      move                 x:(r0)+,x0
end
      move                 a,x:(r1)+
```

Unfortunately, several existing vendor-provided compilers that we tested were not able to exploit this hardware feature of dual data memory banks efficiently; thereby failing to generate highly optimized code for their target ASIPs. This inevitably implies that the users for these ASIPs should hand-optimize their code in assembly to fully exploit dual memory banks, which makes programming the processors quite complex and time consuming.

In this paper, we describe our implementation of two core techniques in the code generation for non-orthogonal ASIPs: *register allocation* and *memory bank assignment*. Our register allocation is decoupled into two phases to handle the heterogeneous register architecture of an ASIP as follows.

1. Physical registers are classified into a set of *register classes*, each of which is a collection of registers dedicated to the same machine instructions; and, our register classification algorithm allocates each temporary variable to one of the register classes.
2. A conventional graph coloring algorithm is slightly modified to assign each temporary a physical register within the register class previously allocated to it.

Our memory bank assignment whose goal is to efficiently assign variables to multi-memory banks for ASIPs is also decoupled into two phases as follows.

1. A *maximum spanning tree* (MST) algorithm is used to find a memory bank assignment for variables.
2. The initial bank assignment by the MST-based algorithm is improved by the graph coloring algorithm that was also used for register assignment.

Our algorithms differ from previous work in that it assigns variables to heterogeneous registers and multi-memory banks in separate, *decoupled* code generation phases, as shown above; while previous work did it in a single, tightly-*coupled* phase [13, 14]. As will be reported later in this paper, our performance results were quite encouraging. First of all, we found that the code generation time was dramatically reduced by a factor of up to four magnitudes of order. This result was somewhat already expected because our decoupled code generation phases greatly simplified the register and bank assignment problem overall. Meanwhile, the benchmarking results also showed that we generated code that is nearly identical in quality to the code generated by the coupled approach in almost every case.

Section 2 discusses the ASIP architecture that we are targeting in this work. Section 3 presents our algorithms, and Section 4 presents our experiments with additional results that we have recently obtained since our earlier preliminary study [5]. Section 5 concludes our discussion.

2. TARGET MACHINE MODEL

In this section, we characterize the non-orthogonal architecture of ASIPs with two properties.

2.1 Heterogeneous Registers

To more formally define this register architecture, we start this section by first presenting the following definitions.

DEFINITION 1. *Given a target machine M, let I = $\{i_1, i_2,$ ..., $i_n\}$ be the set of all the instructions defined on M, and R = $\{r_1, r_2, ..., r_m\}$ be the set of all its registers. For instruction $i_j \in I$, we define the set of all its operands, $Op(i_j) = \{O_{j1}, O_{j2}, ..., O_{jk}\}$. Assume C_{jl} is the set of all the registers that can appear at the position of some operand O_{jl}, $1 \leq l \leq k$. Then we say here that C_{jl} forms a* **register class** *for instruction i_j.*

DEFINITION 2. *From Definition 1, we define S_j, a collection of distinct register classes for instruction i_j, as follows:*

$$S_j = \bigcup_{l=1}^{k} \{C_{jl}\}. \qquad (1)$$

From this, we in turn define S as follows:

$$S = \bigcup_{j=1}^{n} S_j. \qquad (2)$$

We say that S is the whole collection of register classes for machine M.

To see the difference of homogeneous and heterogenous register architectures, first consider the SPARC as an example of a processor with homogenous registers. A typical instruction of the SPARC has three operands

$$\texttt{op\_code} \ reg_i, reg_j, reg_k$$

where all the 32 registers(r0,r1,. . .,r31) in the register file can appear as the first operand reg_i. In this case, the set of all these registers forms a single register class for `op_code`. Since for the other operands reg_j and reg_k, the same 32 registers can appear, they again form the same class for the instruction. Thus, we have only one register class defined for the instruction `op_code`. On the other hand, the DSP56000 has an instruction of the form

$$\texttt{mpya} \ reg_i, reg_j, reg_k$$

which multiplies the first two operands and places the product in the third operand. The DSP56000 restricts reg_i and reg_j to be input registers X0, X1, Y0, Y1, and reg_k to be accumulator A or B. In this case, we have two register classes defined for `mpya`: {X0, X1, Y0, Y1} at reg_i and reg_j and {A,B} at reg_k.

In the above examples, S_j for `op_code` and `mpya` are, respectively, {{r0,. . .,r31}} and {{X0, X1, Y0, Y1},{A,B}}. We say that a typical processor with n general purpose registers like the SPARC has a homogeneous register architecture. This is mainly because S is usually a set of a single element consisting of the n registers for the processor, which, by Definitions 1 and 2, equivalently means that the same n registers are homogeneous in all the machine instructions. In the case of DSP56000, however, its registers are dedicated differently to the machine instructions, which make them partially homogeneous only in the subsets of machine instructions. For example, we can see that even one instruction like `mpya` of DSP56000 has two different sets of homogenous registers: XYN and AB. We list the whole collection of register classes defined for DSP56000 in Table 1. In general, we say that a machine with such complex register classes has a heterogeneous register architecture.

ID	Register Class	Physical Registers
1	XYN	X0, X1, Y0, Y1
2	XY	X, Y (long word)
3	YR	R4 – R7
4	AB	Accumulator A, B
5	YN	N4 – N7
6	XR	R0 – R3
7	XN	N0 – N3
8	X	X0, X1
9	Y	Y0, Y1

Table 1: The register classes for Motorola DSP56000

2.2 Multiple Data Memory Banks

As an example of multi-memory bank ASIPs, we will use the DSP56000 whose data path was shown in Figure 1, where the ALU operations are divided into data operations and address operations. Data ALU operations are performed on a data ALU with *data registers* which consist of four 24-bit input registers (X0, X1, Y0 and Y1) and two 56-bit accumulators (A and B). Address ALU operations are performed in the *address generation unit* (AGU), which calculates memory addresses necessary to indirectly address data operands in memory. Since the AGU operates independently from the data ALU, address calculations can occur simultaneously with data ALU operations.

As shown in Figure 1, the AGU is divided into two identical halves, each of which has an address ALU and two sets of 16-bit register files. One set of the register files has four *address registers* (R0 – R3), and the other also has four address registers (R4 – R7). The address output multiplexers select the source for the XAB, YAB. The source of each effective address may be the output of the address ALU for indexed addressing or an address register for register-indirect addressing. At every cycle, the addresses generated by the ALUs can be used to access two words in parallel in the X and Y memory banks, each of which consists of 512-word $\times$ 24-bit memory.

Possible memory reference modes of the DSP56000 are of four types: X, Y, L and XY. In X and Y memory reference modes, the operand is a single word either from X or Y memory bank. In L memory reference mode, the operand is a long word (two words each from X and Y memories) referenced by one operand address. In XY memory reference mode, two independent addresses are used to move two word operands to memory simultaneously: one word operand is in X memory, and the other word operand is in Y memory. Such independent moves of data in the same cycle are called a *parallel move*. In Figure 1, we can see two data buses XDB and YDB that connect the data path of the DSP56000 to two data memory banks X and Y, respectively. Through these buses, a parallel move is made between memories and data registers.

These architectural features of the DSP56000, like most other ASIPs with multi-memory banks, allow a single instruction to perform one data ALU operation and two move operations in parallel per cycle, but only under certain conditions due to hardware constraints. In the case of the DSP56000, the following *parallel move conditions* should be met to maximize the utilization of the dual memory bank architecture: (1) the two words should be addressed from different memory banks; memory indirect addressing modes using address registers are used to address the words; and, each address register involved in a parallel move must be from a different set among the two register files in the AGU. In this implementation, we attempt to make the parallel move conditions meet in the code so that as many parallel moves as possible can be generated.

3. REGISTER ALLOCATION AND MEMORY BANK ASSIGNMENT

In this section, we detail the code generation phases for register allocation and memory bank assignment, which were briefly described in Section 1. To explain step by step how our code generator produces the final code, we will use the example of DSP56000 assembly code shown in Figure 2. This code can be obtained immediately after the instruction selection phase. Note that it is still in a sequential and unoptimized form. This initial code will be given to the subsequent phases, and optimized for the dual memory architecture of DSP56000, as described in this section.

```
MOVE    a, r0
MOVE    b, r1
MOVE    c, r2
MAC     r0, r1, r2
MOVE    low(r2), low(v)
MOVE    high(r2), high(v)
MOVE    d, r3
MOVE    e, r4
MOVE    f, r5
MAC     r3, r4, r5
MOVE    low(r5), low(w)
MOVE    high(r5), high(w)
```

Figure 2: Example of uncompacted DSP56000 assembly code produced after instruction selection

3.1 Register Class Allocation

In our compiler, instruction selection is decoupled from register allocation and all other subsequent phases, In fact, many conventional compilers such as gcc, lcc and Zephyr that have been targeting GPPs, also separate these two phases. Separating register allocation from instruction selection is relatively straightforward for a compiler targeting GPPs because GPPs have homogeneous registers within a single class, or possibly just a few classes, of registers; that is, in the instruction selection phase, instructions that need registers are assigned symbolic temporaries which, later in the register allocation phase, are mapped to any available registers in the same register class. In ASIPs, however, the register classes for each individual instruction may differ, and a register may belong to many different register classes (see Table 1).

What all this implies is that the relationship between registers and instructions is tightly coupled so that when we select an instruction, somehow we should also determine from which register classes registers are assigned to the instruction. Therefore, *phase-coupling* [9], a technique to cleverly combine these closely related phases, has been the norm for most compilers generating code for ASIPs. However, this phase-coupling may create too many constraints for code generation, thus increasing the compilation time tremendously, as in the case of previous work which will be compared with our approach in Section 4.1.

To relieve this problem in our decoupled approach and still handle a heterogeneous register structure, we implemented a simple scheme that enforces a relationship that binds these two separate phases by inserting another phase, called *register class allocation*, between them. In this scheme, we represent a register in two notions: a register class and a register number in the class. In the register class allocation phase, temporaries are not allocated physical registers, but a set of possible registers (that is, a register class) which can be placed as operands of an instruction. Physical registers are selected among the register class for each instruction in a later phase, which we call *register assignment*. Since the focus of this paper is not on the register class allocation, we cannot discuss the whole algorithm here. Refer to [6] for more details.

The register classes that are allocated for the code in Figure 2 are shown below. They are associated with each temporary $\mathbf{r}i$ referenced in the code.

```
r0 : XYN
r1 : XYN
r2 : AB
r2 : XYN
r2 : XYN
r2 : AB
```

Between register class allocation and register assignment, the *code compaction* phase results in not only reduced code size, but also in exploitation of machine instructions that perform parallel operations, such as the one with an add plus a parallel move. Figure 3 shows the resulting instructions after the code in Figure 2 is compacted. We can see in the compacted code that one MAC (multiply-and-add) and two moves are now combined into a single instruction word, and two moves are combined into one parallel move instruction. We use the traditional *list scheduling* algorithm for our code compaction.

```
MOVE                a,r0  b,r1
MOVE                c,r2  d,r3
MAC  r0, r1, r2     e,r4  f,r5
MAC  r3, r4, r5     low(r2),low(v)
MOVE                high(r2),high(v)  low(r5),low(w)
MOVE                high(r5),high(w)
```

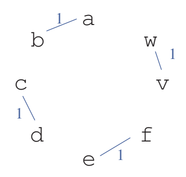
Figure 3: Code sequence after compacting the code in Figure 2

3.2 Memory Bank Assignment

After register class allocation and code compaction, each variable in the resulting code is assigned to one of a set of memory banks (in this example, banks X or Y of the DSP56000). In this section, we present our memory bank assignment technique using two well-known algorithms.

3.2.1 Using a MST Algorithm

In the memory bank assignment phase, we use a MST algorithm. The first step of this basic phase is to construct a weighted undirected graph, which we called the *simultaneous reference graph* (SRG). The graph contains variables referenced in the code as nodes. An edge $e = (v_j, v_k)$ in the SRG means that both variables v_j and v_k are referenced within the same instruction word in the compacted code. Figure 4(a) shows an SRG for the code from Figure 3. The weight on an edge between two variables represents the number of times the variables are referenced within the same word.



(a) SRG

X	Y
a	b
c	d
e	f
v	w

(b) Assigned Memory Bank

```
MOVE                X:a,r0                Y:b,r1
MOVE                X:c,r2                Y:d,r3
MAC  r0, r1, r2     X:e,r4                Y:f,r5
MAC  r3, r4, r5     low(r2),X:low(v)
MOVE                high(r2),X:high(v)    low(r5),Y:low(w)
MOVE                                      high(r5),Y:high(w)
```

(c) Memory Bank Assignment

Figure 4: Code result after memory bank assignment determined from its SRG built for the code in Figure 3

According to the parallel move conditions, two variables referenced in an instruction word must be assigned to different memory banks in order to fetch them in a single instruction cycle. Otherwise, an extra cycle would be needed to access them. Thus, the strategy that we take to maximize the memory throughput is to assign a pair of variables referenced in the same word to different memory banks whenever it is possible. If a conflict occurs between two pairs of variables, the variables in one pair that appear more frequently in the same words shall have a higher priority over those in the other pair. Notice here that the frequency is denoted by the weight in the SRG.

Figure 4(b) shows that the variables a, c, e, and v are assigned to X memory, and the remaining ones b, d, f, and w are to Y memory. This is optimal because all pairs of variables connected via edges are assigned to different memories X and Y, thus avoiding extra cycles to fetch variables, as can be seen from the resulting code in Figure 4(c). In the case of variables v and w, we still need two cycles to move each of them because they are long type variables with double-word length. However, they also benefit from the optimal memory assignment as each half of the variables is moved together in the same cycle.

The memory bank assignment problem that we face in reality is not always as simple as the one in Figure 4. To illustrate a more realistic and complex case of the problem, consider Figure 5 where the SRG has five variables.

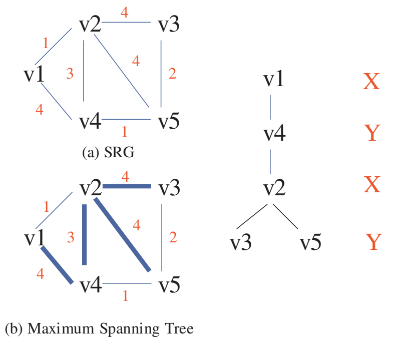


Figure 5: More complex example of a simultaneous reference graph and the maximum spanning tree constructed from it

We view the process of assigning n memory banks as that of dividing the SRG into n disjoint subgraphs; that is, all nodes in the same subgraph are assigned a memory bank that corresponds to the subgraph. In our compiler, therefore, we try to obtain an optimal memory bank assignment for a given SRG by finding a *partition* of the graph with the minimum cost according to Definition 3.

DEFINITION 3. *Let $G = (V, E)$ be a connected, weighted graph where V is a set of nodes and E is a set of edges. Let w_e be the weight on an edge $e \in E$. Suppose that a* **partition** *$P = <G_1, G_2, \cdots, G_n>$ of the graph G divides G into n disjoint subgraphs $G_i = (V_i, E_i)$, $1 \le i \le n$, such that $(v_j, v_k) \in E_i$ if $(v_j, v_k) \in E$ for $v_j \in V_i$ and $v_k \in V_i$. Then, the* **cost** *of the partition P is defined as*

$$\sum_{i=1}^{n} \sum_{e \in E_i} w_e.$$

Finding such an optimal partition with the minimum cost is another NP-complete problem. So, we developed a greedy approximation algorithm with $O(|E|+|V|lg|V|)$ time complexity, as shown in Figure 6. Since in practice $|E| \approx |V|$ for our problem, the algorithm usually runs fast in $O(|V|lg|V|)$ time. In the algorithm, we assume $n = 2$ because virtually no existing ASIPs have more than two data memory banks. But, this algorithm can be easily extended to handle the cases for $n > 2$.

In our memory bank assignment algorithm, we first identify a *maximum spanning tree* (MST) of the SRG. Given a connected graph G, a *spanning tree* of G is a connected acyclic subgraph that covers all nodes of G. A MST is a spanning tree whose total weight of all its edges is not less than those of any other spanning trees of G. One interesting property of a spanning tree is that it is a bipartite graph as any tree is actually bipartite. So, given a spanning tree T for a graph G, we can obtain a partition $P = <G_1, G_2>$ from T by, starting from an arbitrary node, say u, in T, assigning to G_1 all nodes an even distance from u and to G_2 those an odd distance from u.

Based on this observation, our algorithm is designed to first identify a spanning tree from the SRG, and then, to compute a partition from it. But we here use a heuristic that chooses not an ordinary spanning tree but a maximum spanning tree. The rationale for the heuristic is that, if we build a partition from a MST, we can eliminate heavy-weighted edges of the MST, thereby increasing the chance to reduce the overall cost of the resultant partition. Unfortunately, constructing a partition from a MST does not guarantee the optimum solution. But, according to our earlier preliminary work [5], the notion of a MST provides us a crucial idea about how to find a partition with low cost, which is in turn necessary to find a near-optimal memory bank assignment. For instance, our algorithm can find an optimal partitioning for the SRG in Figure 5.

To find a MST, our algorithm uses Prim's MST algorithm [11]. Our algorithm is global; that is, it is applied across basic blocks. For each node, the following sequence is repeatedly iterated until all SRG nodes have been marked. In the algorithm, the edges in the *priority queue* Q are sorted in the order of their weights, and an edge with the highest weight is removed first. When there is more than one edge with the same highest weight, the one that was inserted first will be removed. Note here that the simultaneous reference graph G_{SR} is not necessarily connected, as opposed to our assumption made above. Therefore, we create a set of MSTs one for each connected subgraph of G_{SR}. Also, note in the algorithm that at least one of the nodes w and z should always be marked because the edges of a marked node u was always inserted in Q earlier in the algorithm. Figure 5(b) shows the spanned tree obtained after this algorithm is applied to the SRG given in Figure 5(a). We can see that X memory is assigned in even depth and Y memory in odd depth in this tree.

3.2.2 *Using a Graph Coloring Algorithm*

A graph coloring approach [4] has been traditionally used for register allocation in many compilers. The central idea of graph coloring is to partition each variable into separate live ranges, where each live range is a candidate to be allocated to a register rather than entire variables. We have found that the same idea can be also used to improve the basic memory bank assignment described in Section *3.2.1* by relaxing the name-related constraints on variables that are to be assigned

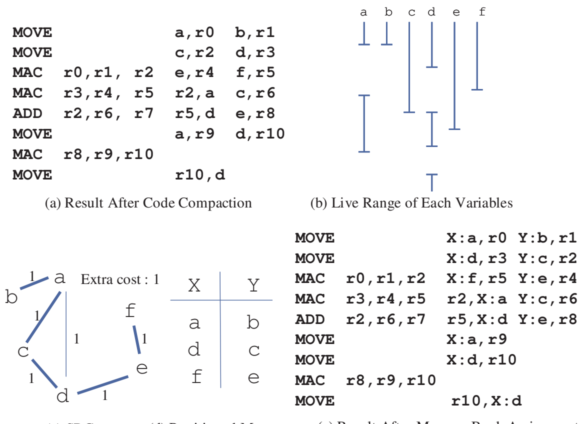
```
Input:    a simultaneous reference graph $G_{SR} = (V_{SR}, E_{SR})$
Output:   a set $V_{SR}$ whose nodes are all colored either with $X$ or $Y$

Algorithm:
    $S_T \leftarrow Q \leftarrow \emptyset$;  // $S_T$ is a set of MSTs and $Q$ is a priority queue
    for all nodes $v$ in $V_{SR}$ do unmark $v$;
    $u \leftarrow$ select_unmarked_node_in($V_{SR}$);
            // Return $\perp$ if every node in $V_{SR}$ is marked
    $i \leftarrow 1$;   create a new MST $T_i$;
    while $u \neq \perp$ do  // Find all MSTs for connected subgraphs of $G_{SR}$
        mark $u$;
        $E_u \leftarrow$ the set of all edges incident on $u$;
        sort the elements of $E_u$ in incresing order
                by weights, and add them to $Q$;
        while $Q \neq \emptyset$ do
            remove an edge $e = (w, z)$ with highest priority from $Q$;
            if $z$ is unmarked then
                $T_i \leftarrow T_i \cup \{e\}$;   $u \leftarrow z$;   break;
            fi
            if $w$ is unmarked then
                $T_i \leftarrow T_i \cup \{e\}$;   $u \leftarrow w$;   break;
            fi
        od
        if $u$ is marked then
         // All nodes in a connected subgraph of $G_{SR}$ have been visited
            $u \leftarrow$ select_unmarked_node_in($V_{SR}$);
            // Select a node in another subgraph, if any, of $G_{SR}$
            add $T_i$ to $S_T$;   $i$++;   create a new MST $T_i$;
        fi
    od
    for all nodes $v$ in $V_{SR}$ do uncolor $v$;
    for every MST $T_i \in S_T$ do
        // Assign variables in $T_i$'s to memory banks $X$ and $Y$
        $next\_visitors\_Q \leftarrow \emptyset$;
        $m \leftarrow$ # of nodes in $V_{SR}$ of $X$-color
              $-$ # of nodes in $V_{SR}$ of $Y$-color;
        select an arbitrary node $v$ in $T_i$;
        if $m > 0$ then                    // More nodes have been X-colored
            color $v$ with $Y$-color;
        else                               // More nodes have been Y-colored
            color $v$ with $X$-color;
        repeat
            for every node $u$ adjacent to $v$ do
                if $u$ is not colored then
                    color $u$ with a color different from the color of $v$;
                    append $u$ to $next\_visitors\_Q$;
                fi
            $v \leftarrow$ extract one node from $next\_visitors\_Q$;
        until all nodes in $T_i$ are colored;
    od
    $m \leftarrow$ # of nodes in $V_{SR}$ of $X$-color
          $-$ # of nodes in $V_{SR}$ of $Y$-color;
    while $m > 0$ do
     // While there are more $X$-colored nodes than $Y$-colored ones
        if $\exists$ uncolored node $v \in V_{SR}$ then
            color $v$ with $Y$-color;   $m$- -;
        else break;
    while $m < 0$ do
     // While there are more $Y$-colored nodes than $Y$-colored ones
        if $\exists$ uncolored node $v \in V_{SR}$ then
            color $v$ with $X$-color;   $m$++;
        else break;
    if $m = 0$ then
        for any uncolored node $v$ in $V_{SR}$ do
            color $v$ alternately with $X$ and $Y$ colors;
    return $V_{SR}$;
```

Figure 6: A memory bank assignment algorithm for dual memories X and Y

to memory banks.

In this approach, we build an undirected graph, called the *memory bank interference graph*, to determine which live ranges conflict and could not be assigned to the same memory bank. Disjoint live ranges of the same variable can be assigned to different memory banks after giving a new name to each live range. This additional flexibility of a graph coloring approach can sometimes result in a more efficient allocation of variables to memory banks, as we will show in this section.

Two techniques, called *name splitting* and *merging*, have been newly implemented to help the memory bank assignment benefit from this graph coloring approach. The example in Figure 4 is too simple to illustrate this; hence, let us consider another example in Figure 7 that will serve to clarify various features of these techniques.



```
MOVE                  a,r0  b,r1
MOVE                  c,r2  d,r3
MAC   r0,r1, r2       e,r4  f,r5
MAC   r3,r4, r5       r2,a  c,r6
ADD   r2,r6, r7       r5,d  e,r8
MOVE                  a,r9  d,r10
MAC   r8,r9,r10
MOVE                  r10,d
```

(a) Result After Code Compaction

(b) Live Range of Each Variables

(c) SRG

X	Y
a	b
d	c
f	e

(d) Partitioned Memory

```
MOVE                  X:a,r0 Y:b,r1
MOVE                  X:d,r3 Y:c,r2
MAC   r0,r1,r2        X:f,r5 Y:e,r4
MAC   r3,r4,r5        r2,X:a Y:c,r6
ADD   r2,r6,r7        r5,X:d Y:e,r8
MOVE                  X:a,r9
MOVE                  X:d,r10
MAC   r8,r9,r10
MOVE                   r10,X:d
```

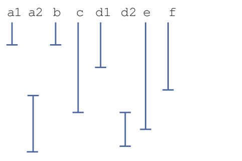
(e) Result After Memory Bank Assignment

Figure 7: Code example and data structures to illustrate name splitting and merging

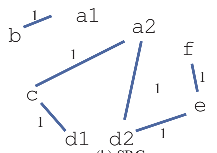
Figure 7(a) shows an example of code that is generated after code compaction, and Figure 7(b) depicts the live ranges of each of the variables. Note that the variables a and d each have multiple live ranges. Figures 7(c) and 7(d) show the SRG and the assignment of variables to memory banks. We can see that a single parallel move cannot be exploited in the example because a and d were assigned to the same data memory. Finally, Figure 7(e) shows the resulting code after memory banks are assigned by using the MST algorithm with the memory partitioning information from Figure 7(d).

Figure 8 shows how name splitting can improve the same example in Figure 7. Name splitting is a technique that tries to reduce the code size by compacting more memory references into parallel move instructions. This technique is based on a well-known graph coloring approach. Therefore, instead of presenting the whole algorithm, we will describe the technique with an example given in Figure 8. We can see in Figure 8(a) that each live range of the variable is a candidate for being assigned to a memory bank. In the example, the two variables a and d with disjoint live ranges are *split*; that is, each live range of the variables are given different names.

Figures 8(b) and 8(c) show the modified SRG and the improved assignment of variables to memory banks. Figure 8(d) demonstrates that, by considering live ranges as opposed to entire variables for bank assignment, we can place the two live



(a) Live Range After Local Variable Renaming



(b) SRG

X	Y
a1	b
a2	c
d1	d2
e	f

(c) Partitioned Memory

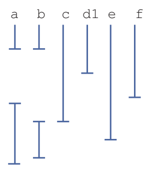
```
MOVE                X:a1,r0 Y:b,r1
MOVE                X:d1,r3 Y:c,r2
MAC   r0,r1,r2      X:f,r5  Y:e,r4
MAC   r3,r4,r5      r2,X:a2 Y:c, r6
ADD   r2,r6,r7      r5,X:d2 Y:e,r8
MOVE                X:a2,r9 Y:d2,r10
MAC   r8,r9,r10
MOVE                r10,X:d2
```

(d) Result After Name Splitting

Figure 8: Name splitting for local variables

ranges of d in different memory banks, which allows us to exploit a parallel move after eliminating one MOVE instruction from the code in Figure 7(e).

Although name splitting helps us to further reduce the code size, it may increase the data space, as we monitored in Figure 8. To mitigate this problem, we *merge* names after name splitting. Figure 9 shows how the data space for the same example can be improved using name merging. In the earlier example, we split a into two names a1 and a2 according to the live ranges for a, and these new names were assigned to the same memory bank. Note that these live ranges do not conflict. This means that they can in turn be assigned to the same location in memory.



(a) Live Range After Local Variable Merging

X	Y
a	b(d2)
d1	c
e	f

(b) Partitioned Memory

Figure 9: Name merging for local variables

Not only can the compiler merge nonconflicting live ranges of the same variable, as in the case of the variable a, but it can also merge nonconflicting live ranges of different variables. We see in Figure 9(b) that two names b and d2 are merged to save one word in Y memory.

The key idea of name splitting and merging is to consider live ranges, instead of entire variables, as candidates to be assigned to memory banks. As can be seen in above examples, the compiler can potentially reduce both the number of executed instructions by exploiting parallel moves and the number of memory words required.

We have shown in this work that applying graph coloring techniques when assigning variables to memory banks has a greater potential for improvement than applying these techniques for register assignment. The reason is that the number of memory banks is typically much smaller than the number of registers; thereby, the algorithm for name splitting and merging has practically polynomial time complexity even though name splitting and merging basically use a theoretically NP-complete graph coloring algorithm. That is, asymptotically the time required for name splitting and merging scales at worst case as $n2^n$ for dual data memory banks. This is yet much faster than conventional graph coloring for register allocation, whose time complexity is $O(nm^n)$ where m is typically more than 32 for GPPs. It has already been empirically proven that in practice, register allocation with such high complexity runs in polynomial time thanks to numerous heuristics such as pruning. So does name splitting and merging, as we will demonstrate in Section 4.

3.3 Register Assignment

After memory banks are determined for each variable in the code, physical registers are assigned to the code. For this, we again use the graph coloring algorithm with special constraints added to handle non-orthogonal architectures. To explain these constraints, recall that we only allocated register classes to temporaries earlier in the register class allocation phase. For register assignment, each temporary is assigned one physical register among those in the register class allocated to the temporary. For example, the temporary $r0$ in Figure 4 shall be replaced by one register among four candidates $X0$, $X1$, $Y0$ and $Y1$, because Table 1 indicates that they are in register class 1, which is currently allocated to $r0$ as shown in Section 3.1.

In addition to register class constraints, register assignment also needs to consider additional constraints for certain types of instructions. For instance, register assignment for instructions containing a parallel move, such as those in Figure 4, must meet the following architectural constraints on dual memory banks: data from each memory bank should be moved to a predefined set of registers. This constraint is also due partially to the heterogeneous register architecture of ASIPs. Back in the example from Figure 4, the variable a in the parallel move with $r0$ is allocated to memory X. Therefore, only registers eligible for $r0$ are confined to $X0$ and $X1$. If these physical registers are already assigned to other instructions, then a register spill will occur.

Satisfying all these constraints on register classes and memory banks, our graph coloring algorithm assigned temporaries to physical registers in the code. Figure 10 shows the resulting code after register assignment is applied to the code shown in Figure 4(c). We can see in Figure 10 that memory references in the code represented symbolically in terms of variable names like a and b are now converted into real ones using addressing modes provided in the machine. This conversion was done in the memory offset assignment phase that comes after register assignment. In this final phase, we applied an algorithm similar to the *maximum weighted path* (MWP) algorithm originally proposed by Leupers and Marwede [7].

```
MOVE                X:(r1)+,X0      Y:(r5)+,Y0
MOVE                X:(r1)+,A       Y:(r5)+,Y1
MAC   X0, Y0, A     X:(r1)+,X1      Y:(r5)+,B
MAC   X1, Y1, B     A0,X:(r1)+
MOVE                A1,X:(r1)       B0,Y:(r5)+
MOVE                                B1,Y:(r5)
```

Figure 10: Resulting code after register assignment and memory offset assignment

4. COMPARATIVE EMPIRICAL STUDIES

To evaluate the performance of our memory bank assignment algorithm, we implemented the algorithm and conducted experiments with benchmark suites on a DSP56000 [10]. The performance is measured in two metrics: size and time. In this section, we report the performance obtained in our experiments, and compare our results with other work.

4.1 Comparison with Previous Work

Not until recently had code generation for ASIPs received much attention from the main stream of conventional compiler research. One prominent example of a compiler study targeting ASIPs may be that of Araujo and Malik [2] who proposed a linear-time optimal algorithm for instruction selection, register allocation, and instruction scheduling for expression trees. Like most other previous studies for ASIPs, their algorithm was not designed specifically for the multi-memory bank architectures. To the best of our knowledge, the earliest study that addressed this problem of *register* and *memory bank assignment* is that of Saghir et al. [12]. However, our work differs from theirs because we target ASIPs with heterogeneous registers while theirs assume processors with a large number of centralized general-purpose registers. By the same token, our approach also differs from the *RAW* project at MIT [3] since their memory bank assignment techniques neither assume heterogeneous registers. nor even ASIPs.

Most recently, this problem was extensively addressed in a project, called *SPAM*, conducted by researchers at Princeton and MIT [1, 14]. In fact, SPAM is the only closely related work that is currently available to us. Therefore, in this work, we compared our algorithm with theirs by experimenting with the same set of benchmarks targeting the same processor.

4.2 Comparison of Code Size

In Figure 11, we list the benchmarks that were compiled by both the SPAM compiler and ours. These benchmarks are from the *ADPCM* and *DSPStone* [15] suites. For some reason, we could not port SPAM successfully on our machine platform. So, the numbers for SPAM in the figure are borrowed from their literature [14] in a comparison with our experimental result.

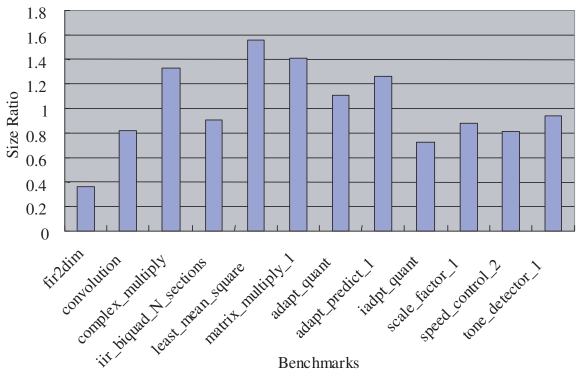


Figure 11: **Ratios of our code sizes to SPAM code sizes**

The figure displays the size ratios of our code to SPAM code; that is, SPAM code size is 1 and our code size is normalized against SPAM code size. In the figure, we can see that the sizes of our output code are comparable to those of their code overall. In fact, for seven benchmarks out of the twelve, our output code is smaller than SPAM code. These results indicate that our memory bank assignment algorithm is as effective as their simultaneous reference allocation algorithm in most cases.

4.3 Comparison of Compilation Time

While both compilers demonstrate comparable performance in code size, the difference of compilation times is significant, as depicted in Figure 12. According to their literature [14], all experiments of SPAM were conducted on Sun Microsystems Ultra Enterprise featuring eight processors and 1GB RAM. Unfortunately, we could not find exactly the same machine that they used. Instead, we experimented on the same Sun Microsystems Ultra Enterprise but with two processors and 2GB RAM.

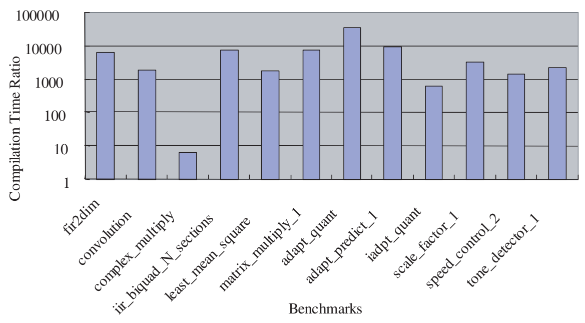


Figure 12: **Ratios (in log scale) of compilation times of our compiler to those of the SPAM compiler**

We can see in the figure that our compilation times were roughly three to four orders of magnitude faster. Despite the differences of machine platforms, therefore, we believe that such large difference of compilation times clearly demonstrates the advantage of our approach over theirs in terms of compilation speed.

Our comparative experiments show evidence that the compilation time of SPAM may increase substantially for large applications, as opposed to ours. We have found that the long compilation time in the SPAM compiler results from the fact that they use a *coupled* approach that attempts to deal with register and memory bank assignment in a single, combined step, where several code generation phases are coupled and simultaneously considered to address the issue. That is, in their approach, variables are allocated to physical registers at the same time they are assigned the memory banks.

To support their coupled approach, they build a *constraint graph* that represents multiple constraints under which an optimal solution to their problem is sought. Unfortunately, these multiple constraints in the graph turn their problem into a typical multivariate optimum problem which is tractable only by an NP-complete algorithm. In this coupled approach, multivariate constraints are unavoidable as various constraints on many heterogeneous registers and multi-memory banks should be all involved to find an optimal reference allocation simultaneously. As a consequence, to avoid using such an expensive algorithm, they inevitably resorted to a heuristic algorithm, called *simulated annealing*, based on a Monte Carlo approach. However, even with this heuristic, we have observed from their literature [13, 14] that their compiler still had to take more than 1000 seconds even for a moderately sized program. This is mainly because the number of constraint in their constraint graph rapidly becomes too large and complicated as the code size increases.

We see that the slowdown in compilation is obviously caused by the intrinsic complexity of their coupled approach. In contrast, our compilation times stayed short even for larger benchmarks. We credit this mainly to our *decoupled* approach which facilitated our application of various fast heuristic algorithms that individually conquer each subproblem encountered in the code generation process for the dual memory bank system. More specifically, in our approach, register allocation is decoupled from code compaction and memory bank assignment; thereby, the binding of physical registers to temporaries comes only after code has been compacted and variables assigned to memory banks.

Some could initially expect a degradation of our output code quality due to the limitations newly introduced by considering physical register binding separately from memory bank assignment. However, we conclude from these results that careful decoupling may alleviate such drawbacks in practice while maximizing the advantages in terms of compilation speed, which is often a critical factor for industry compilers.

4.4 Comparison of Execution Speed

To estimate the impact of code size reduction on the running time, we generated three versions of the code as follows.

uncompacted The first version is our uncompacted code, such as shown in Figure 2, generated immediately after the instruction selection phase.

compiler-optimized The uncompacted code is optimized for DSP56000 by using the techniques in Section 3 to produce the code like the one in Figure 10.

hand-optimized The uncompacted code is optimized by hand. We hand-optimized the same code that the compiler used as the input so that the hand-optimized one may provide us with the upper limit of the performance of the benchmarks on DSP56000.

Their execution times are compared in Figure 13 where the ratios of speedup improvement produced by both compiler-optimization and hand-optimization compared to the speedup produced by the uncompacted code. For instance, the compiler-optimized code for `complex_multiply` achieves speedup of about 23% over the uncompacted code while the hand-optimized code achieves additional speedup of 9%, which is tantamount to 32% in total over the uncompacted code.

In Figure 13, we can see that the average speedup of our compiler-optimized code over the uncompacted code is about 7%, and that of hand-optimized code over the compiler-optimized code is 8%. These results indicate that the compiler has achieved roughly the half of the speedup we could get by hand optimization. Although these numbers may not be satisfactory, the results also indicate that, in six benchmarks out of the twelve, our compiler has achieved the greater part of the performance gains achieved by hand optimization.

Of course, we also have several benchmarks, such as `fir2dim`, `convolution` and `least_mean_square`, in which our compiler has much room for improvement. According to our analysis, the main cause that creates such difference in execution

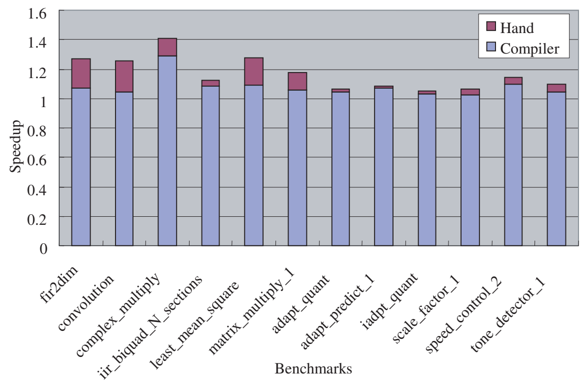


Figure 13: Speedups of the execution times of both compiler-optimized and hand-optimized code over the execution time of unoptimized code

time between the compiler-generated code and the hand optimized code is the incapability of our compiler to efficiently handle loops. To illustrate this, consider the example in Figure 14, which shows a typical example where software pipelining is required to optimize the loop.

```
    DO  #16, L10
    MOV          X:a,X0 Y:b,Y0
    MPY X0,Y0,A  X:c,X1 Y:d,Y1
…
    ADD X1,Y1,A
    MOV          A,X:e
L10
```

(a) Compiled Compacted Code by Our Approach

```
    MOV          X:a,X0 Y:b,Y0
    DO  #15, L10
    MPY X0,Y0,A  X:c,X1 Y:d,Y1
…
    ADD X1,Y1,A  X:a,X0 Y:b,Y0
    MOV          A,X:e
L10
…
```

(b) Hand-Optimized Compacted Code

Figure 14: Compaction Difference Between Our Compiled Code and Hand-Optimized Code

Notice in the example that a parallel move for variables `a` and `b` cannot be compacted into the instruction word containing `ADD` because there is a dependence between `MPY` and them. However, after placing one copy of the parallel move into the preamble of the loop, we can now merge the move with `ADD`. Although this optimization may not reduce the total code size, it eliminates one instruction within the loop, which undoubtedly would reduce the total execution time noticeably.

This example informs us that, since most of the execution time is spent in loops, our compiler cannot match hand optimization in run time speed without more advanced loop optimizations, such as software pipelining, based on rigorous dependence analysis. Currently, this issue remains for our future research.

5. SUMMARY AND CONCLUSION

In this paper, we proposed a decoupled approach for supporting a dual memory architecture, where the six code generation phases are performed separately. We also presented *name splitting* and *merging* as additional techniques. By comparing our work with SPAM, we analyzed the pros and cons of our decoupled approach as opposed to their coupled approach. The comparative analysis of the experiments revealed that our compiler achieved comparable results in code size; yet, our decoupled structure of code generation simplified our data allocation algorithm for dual memory banks, which allows the algorithm to run reasonably fast. The analysis also revealed that exploiting dual memory banks by carefully assigning scalar variables to the banks brought about the speedup at run time.

However, the analysis exposed several limitations of the current techniques as well. For instance, while our approach was limited to only scalar variables, we expect that memory bank assignment for arrays can achieve a large performance enhancement because most computations are performed on arrays in number crunching programs. This is actually illustrated in Figures 11 and 13, where even highly hand-optimized code could not make a significant performance improvement in terms of speed although we made a visible difference in terms of size. This is mainly because the impact of scalar variables on the performance is relatively low as compared with the space they occupy in the code. Another limitation would be to perform memory bank assignment on arguments passed via memory to functions. This would require interprocedural analysis since the caller must know the memory access patterns of the callee for passing arguments. Also, certain loop optimization techniques, like those listed in Section 4.4, need to be implemented to further improve execution time of the output code.

6. REFERENCES

[1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. *Challenges in Code Generation for Embedded Processors*, pages 48–64. In Marwedel and Goossens [9], 1995.

[2] G. Araujo and S. Malik. Code Generation for Fixed-point DSPs. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):136–161, April 1998.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computers*, Nov. 2001.

[4] G. Chaitan. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 201–207, June 1982.

[5] J. Cho, J. Kim, and Y. Paek. Efficient and Fast Allocation of On-chip Dual Memory Banks. In *6th Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2002.

[6] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84–92, Nov. 2001.

[7] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Internaltional Conference on Computer-Aided Design*, 1996.

[8] C. Liem. *Retargetable Compilers for Embedded Core Processors*. Kluwer Academic Publishers, 1997.

[9] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[10] Motorola Inc., Austin, TX. *DSP56000 24-Bit Digital signal Processor Family Manual*, 1995.

[11] R. Prim. Shortest Connection Networks and Some Generalizations. *Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[12] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGOPS Operating Systems*, pages 234–243, 1996.

[13] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. PhD thesis, Princeton University Department of EE, May 15, 1998.

[14] A. Sudarsanam and S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):242–264, April 2000.

[15] V. Zivoljnovic, J.M. Velarde, C. Schager, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.